Minimal Maxima

Robert Dodier

September 10, 2005

1 What is Maxima?

Maxima[1] is a system for working with expressions, such as $x+y$, $sin(a+b\pi)$, and $u \cdot v - v \cdot u$.

Maxima is not much worried about the meaning of an expression. Whether an expression is meaningful is for the user to decide.

Sometimes you want to assign values to the unknowns and evaluate the expression. Maxima is happy to do that. But Maxima is also happy to postpone assignment of specific values; you might carry out several manipulations of an expression, and only later (or never) assign values to unknowns.

Let's see some examples.

1. I want to calculate the volume of a sphere.

```
(%i1) V: 4/3 * %pi * r^3;
                                                  3
                                           4 %pi r
(%o1)                                      --------
                                              3
```

2. The radius is 10.

```
(%i2) r: 10;
(%o2)                                         10
```

[1] Home page: http://maxima.sourceforge.net
Documents: http://maxima.sourceforge.net/docs.shtml
Reference manual: http://maxima.sourceforge.net/docs/manual/en/maxima.html

3. V is the same as before; Maxima won't change V until I tell it to do so.

```
(%i3) V;
                                        3
                                  4 %pi r
(%o3)                             --------
                                     3
```

4. Please re-evaluate V, Maxima.

```
(%i4) ''V;
                                  4000 %pi
(%o4)                             --------
                                     3
```

5. I'd like to see a numerical value instead of an expression.

```
(%i5) ''V, numer;
(%o5)                         4188.79020478639
```

2 Expressions

Everything in Maxima is an expression, including mathematical expressions, objects, and programming constructs. An expression is either an atom, or an operator together with its arguments.

An atom is a symbol (a name), a string enclosed in quotation marks, or a number (integer or floating point).

All nonatomic expressions are represented as $op(a_1, \ldots, a_n)$ where op is the name of an operator and $a_1, \ldots, a_n$ are its arguments. (The expression may be displayed differently, but the internal representation is the same.) The arguments of an expression can be atoms or nonatomic expressions.

Mathematical expressions have a mathematical operator, such as $+ - * / < = >$ or a function evaluation such as $\mathbf{sin}(x)$, $\mathbf{bessel_j}(n, x)$. In such cases, the operator is the function.

Objects in Maxima are expressions. A list $[a_1, \ldots, a_n]$ is an expression $\mathbf{list}(a_1, \ldots, a_n)$. A matrix is an expression

$$\mathbf{matrix}(\mathbf{list}(a_{1,1}, \ldots, a_{1,n}), \ldots, \mathbf{list}(a_{m,1}, \ldots, a_{m,n}))$$

Programming constructs are expressions. A code block **block**$(a_1, \ldots, a_n)$ is an expression with operator **block** and arguments $a_1, \ldots, a_n$. A conditional statement **if** a **then** b **elseif** c **then** d is an expression **if**(a, b, c, d). A loop **for** a **in** L **do** S is an expression similar to **do**(a, L, S).

The Maxima function **op** returns the operator of a nonatomic expression. The function **args** returns the arguments of a nonatomic expression. The function **atom** tells whether an expression is an atom.

Let's see some more examples.

1. Atoms are symbols, strings, and numbers. I've grouped several examples into a list so we can see them all together.

```
(%i2) [a, foo, foo_bar, "Hello, world!", 42, 17.29];
(%o2)      [a, foo, foo_bar, Hello, world!, 42, 17.29]
```

2. Mathematical expressions.

```
(%i1) [a + b + c, a * b * c, foo = bar, a*b < c*d];
(%o1)        [c + b + a, a b c, foo = bar, a b < c d]
```

3. Lists and matrices. The elements of a list or matrix can be any kind of expression, even another list or matrix.

```
(%i1) L: [a, b, c, %pi, %e, 1729, 1/(a*d - b*c)];
                                                 1
(%o1)           [a, b, c, %pi, %e, 1729, ---------]
                                         a d - b c
(%i2) L2: [a, b, [c, %pi, [%e, 1729], 1/(a*d - b*c)]];
                                                  1
(%o2)          [a, b, [c, %pi, [%e, 1729], ---------]]
                                          a d - b c
(%i3) L [7];
                                   1
(%o3)                          ---------
                               a d - b c
(%i4) L2 [3];
                                                1
(%o4)              [c, %pi, [%e, 1729], ---------]
                                        a d - b c
(%i5) M: matrix ([%pi, 17], [29, %e]);
```

```
                             [ %pi  17 ]
(%o5)                        [         ]
                             [ 29   %e ]
(%i6) M2: matrix ([[%pi, 17], a*d - b*c], [matrix ([1, a], [b, 7]), %e]);
                     [ [%pi, 17]  a d - b c ]
                     [                      ]
(%o6)                [ [ 1  a ]             ]
                     [ [      ]      %e     ]
                     [ [ b  7 ]             ]
(%i7) M [2][1];
(%o7)                           29
(%i8) M2 [2][1];
                              [ 1  a ]
(%o8)                         [      ]
                              [ b  7 ]
```

4. Programming constructs are expressions. $x : y$ means assign y to x; the value of the assignment expression is y. **block** groups several expressions, and evaluates them one after another; the value of the block is the value of the last expression.

```
(%i1) (a: 42) - (b: 17);
(%o1)                           25
(%i2) [a, b];
(%o2)                        [42, 17]
(%i3) block ([a], a: 42, a^2 - 1600) + block ([b], b: 5, %pi^b);
                                 5
(%o3)                         %pi  + 164
(%i4) (if a > 1 then %pi else %e) + (if b < 0 then 1/2 else 1/7);
                                      1
(%o4)                           %pi + -
                                      7
```

5. **op** returns the operator, **args** returns the arguments, **atom** tells whether an expression is an atom.

```
(%i1) op (p + q);
(%o1)                            +
(%i2) op (p + q > p*q);
(%o2)                            >
(%i3) op (sin (p + q));
(%o3)                           sin
```

```
(%i4) op (foo (p, q));
(%o4)                           foo
(%i5) op (foo (p, q) := p - q);
(%o5)                           :=
(%i6) args (p + q);
(%o6)                         [q, p]
(%i7) args (p + q > p*q);
(%o7)                      [q + p, p q]
(%i8) args (sin (p + q));
(%o8)                         [q + p]
(%i9) args (foo (p, q));
(%o9)                        [p, - q]
(%i10) args (foo (p, q) := p - q);
(%o10)                  [foo(p, q), p - q]
(%i11) atom (p);
(%o11)                          true
(%i12) atom (p + q);
(%o12)                          false
(%i13) atom (sin (p + q));
(%o13)                          false
```

6. Operators and arguments of programming constructs. The single quote tells Maxima to construct the expression but postpone evaluation. We'll come back to that later.

```
(%i1) op ('(block ([a], a: 42, a^2 - 1600)));
(%o1)                          block
(%i2) op ('(if p > q then p else q));
(%o2)                           if
(%i3) op ('(for x in L do print (x)));
(%o3)                          mdoin
(%i4) args ('(block ([a], a: 42, a^2 - 1600)));
                                      2
(%o4)                 [[a], a : 42, a  - 1600]
(%i5) args ('(if p > q then p else q));
(%o5)                  [p > q, p, true, q]
(%i6) args ('(for x in L do print (x)));
(%o6)     [x, L, false, false, false, false, print(x)]
```

3 Evaluation

The value of a symbol is an expression associated with the symbol. Every symbol has a value; if not otherwise assigned a value, a symbol evaluates to itself. (E.g., x evaluates to x if not otherwise assigned a value.)

Numbers and strings evaluate to themselves.

A nonatomic expression is evaluated approximately as follows.

1. Each argument of the operator of the expression is evaluated.
2. If an operator is associated with a callable function, the function is called, and the return value of the function is the value of the expression.

Evaluation is modified in several ways. Some modifications cause less evaluation:

1. Some functions do not evaluate some or all of their arguments, or otherwise modify the evaluation of their arguments.
2. A single quote $'$ prevents evaluation.
   (a) $'a$ evaluates to a. Any other value of a is ignored.
   (b) $'f(a_1, \ldots, a_n)$ evaluates to $f(\mathbf{ev}(a_1), \ldots, \mathbf{ev}(a_n))$. That is, the arguments are evaluated but f is not called.
   (c) $'(\ldots)$ prevents evaluation of any expressions inside (...).

Some modifications cause more evaluation:

1. Two single quotes $''a$ causes an extra evaluation at the time the expression a is parsed.
2. $\mathbf{ev}(a)$ causes an extra evaluation of a every time $\mathbf{ev}(a)$ is evaluated.
3. The idiom $\mathbf{apply}(f, [a_1, \ldots, a_n])$ causes the evaluation of the arguments $a_1, \ldots, a_n$ even if f ordinarily quotes them.
4. **define** constructs a function definition like :=, but **define** evaluates the function body while := quotes it.

Let's consider how some expressions are evaluated.

1. Symbols evaluate to themselves if not otherwise assigned a value.

```
(%i1) block (a: 1, b: 2, e: 5);
(%o1)                          5
(%i2) [a, b, c, d, e];
(%o2)                   [1, 2, c, d, 5]
```

2. Arguments of operators are ordinarily evaluated (unless evaluation is prevented one way or another).

```
(%i1) block (x: %pi, y: %e);
(%o1)                         %e
(%i2) sin (x + y);
(%o2)                     - sin(%e)
(%i3) x > y;
(%o3)                     %pi > %e
(%i4) x!;
(%o4)                        %pi!
```

3. If an operator corresponds to a callable function, the function is called (unless prevented). Otherwise evaluation yields another expression with the same operator.

```
(%i1) foo (p, q) := p - q;
(%o1)                 foo(p, q) := p - q
(%i2) p: %phi;
(%o2)                        %phi
(%i3) foo (p, q);
(%o3)                     %phi - q
(%i4) bar (p, q);
(%o4)                   bar(%phi, q)
```

4. Some functions quote their arguments. Examples: **save**, :=, **kill**.

```
(%i1) block (a: 1, b: %pi, c: x + y);
(%o1)                        y + x
(%i2) [a, b, c];
(%o2)                   [1, %pi, y + x]
(%i3) save ("tmp.save", a, b, c);
(%o3)                      tmp.save
(%i4) f (a) := a^b;
                                   b
```

```
(%o4)                          f(a) := a
(%i5) f (7);
                                  %pi
(%o5)                            7
(%i6) kill (a, b, c);
(%o6)                            done
(%i7) [a, b, c];
(%o7)                          [a, b, c]
```

5. A single quote prevents evaluation even if it would ordinarily happen.

```
(%i1) foo (x, y) := y - x;
(%o1)                      foo(x, y) := y - x
(%i2) block (a: %e, b: 17);
(%o2)                             17
(%i3) foo (a, b);
(%o3)                          17 - %e
(%i4) foo ('a, 'b);
(%o4)                            b - a
(%i5) 'foo (a, b);
(%o5)                        foo(%e, 17)
(%i6) '(foo (a, b));
(%o6)                         foo(a, b)
```

6. Two single quotes (quote-quote) causes an extra evaluation at the time the expression is parsed.

```
(%i1) diff (sin (x), x);
(%o1)                           cos(x)
(%i2) foo (x) := diff (sin (x), x);
(%o2)                 foo(x) := diff(sin(x), x)
(%i3) foo (x) := ''(diff (sin (x), x));
(%o3)                     foo(x) := cos(x)
```

7. **ev** causes an extra evaluation every time it is evaluated. Contrast this with the effect of quote-quote.

```
(%i1) block (xx: yy, yy: zz);
(%o1)                             zz
(%i2) [xx, yy];
(%o2)                          [yy, zz]
(%i3) foo (x) := ''x;
```

```
(%o3)                          foo(x) := x
(%i4) foo (xx);
(%o4)                               yy
(%i5) bar (x) := ev (x);
(%o5)                        bar(x) := ev(x)
(%i6) bar (xx);
(%o6)                               zz
```

8. **apply** causes the evaluation of arguments even if they are ordinarily quoted.

```
(%i1) block (a: aa, b: bb, c: cc);
(%o1)                               cc
(%i2) block (aa: 11, bb: 22, cc: 33);
(%o2)                               33
(%i3) [a, b, c, aa, bb, cc];
(%o3)                   [aa, bb, cc, 11, 22, 33]
(%i4) apply (kill, [a, b, c]);
(%o4)                              done
(%i5) [a, b, c, aa, bb, cc];
(%o5)                   [aa, bb, cc, aa, bb, cc]
(%i6) kill (a, b, c);
(%o6)                              done
(%i7) [a, b, c, aa, bb, cc];
(%o7)                     [a, b, c, aa, bb, cc]
```

9. **define** evaluates the body of a function definition.

```
(%i1) integrate (sin (a*x), x, 0, %pi);
                             1   cos(%pi a)
(%o1)                        - - ----------
                             a       a
(%i2) foo (x) := integrate (sin (a*x), x, 0, %pi);
(%o2)         foo(x) := integrate(sin(a x), x, 0, %pi)
(%i3) define (foo (x), integrate (sin (a*x), x, 0, %pi));
                                  1   cos(%pi a)
(%o3)                 foo(x) := - - ----------
                                  a       a
```

4 Simplification

After evaluating an expression, Maxima attempts to find an equivalent expression which is "simpler." Maxima applies several rules which embody conventional notions of simplicity. For example, $1 + 1$ simplifies to 2, $x + x$ simplifies to $2x$, and **sin**(%**pi**) simplifies to 0.

However, many well-known identities are not applied automatically. For example, double-angle formulas for trigonometric functions, or rearrangements of ratios such as $a/b + c/b \to (a + c)/b$. There are several functions which can apply identities.

Simplification is always applied unless explicitly prevented. Simplification is applied even if an expression is not evaluated.

tellsimpafter establishes user-defined simplification rules.

Let's see some examples of simplification.

1. Quote mark prevents evaluation but not simplification. When the global flag **simp** is **false**, simplification is prevented but not evaluation.

```
(%i1) '[1 + 1, x + x, x * x, sin (%pi)];
                                          2
(%o1)                        [2, 2 x, x , 0]
(%i2) simp: false$
(%i3) block ([x: 1], x + x);
(%o3)                            1 + 1
```

2. Some identities are not applied automatically. **expand**, **ratsimp**, **trigexpand**, **demoivre** are some functions which apply identities.

```
(%i1) (a + b)^2;
                                          2
(%o1)                          (b + a)
(%i2) expand (%);
                             2               2
(%o2)                        b  + 2 a b + a
(%i3) a/b + c/b;
                                   c   a
(%o3)                              - + -
                                   b   b
```

```
(%i4) ratsimp (%);
                                     c + a
(%o4)                                -----
                                       b
(%i5) sin (2*x);
(%o5)                              sin(2 x)
(%i6) trigexpand (%);
(%o6)                           2 cos(x) sin(x)
(%i7) a * exp (b * %i);
                                       %i b
(%o7)                              a %e
(%i8) demoivre (%);
(%o8)                       a (%i sin(b) + cos(b))
```

5 apply, map, and lambda

1. **apply** constructs and evaluates an expression. The arguments of the expression are always evaluated (even if they wouldn't be otherwise).

```
(%i1) apply (sin, [x * %pi]);
(%o1)                             sin(%pi x)
(%i2) L: [a, b, c, x, y, z];
(%o2)                          [a, b, c, x, y, z]
(%i3) apply ("+", L);
(%o3)                        z + y + x + c + b + a
```

2. **map** constructs and evaluates an expression for each item in a list of arguments. The arguments of the expression are always evaluated (even if they wouldn't be otherwise). The result is a list.

```
(%i1) map (foo, [x, y, z]);
(%o1)                     [foo(x), foo(y), foo(z)]
(%i2) map ("+", [1, 2, 3], [a, b, c]);
(%o2)                       [a + 1, b + 2, c + 3]
(%i3) map (atom, [a, b, c, a + b, a + b + c]);
(%o3)                 [true, true, true, false, false]
```

3. **lambda** constructs a lambda expression (i.e., an unnamed function). The lambda expression can be used in some contexts like an ordinary named function. **lambda** does not evaluate the function body.

```
(%i1) f: lambda ([x, y], (x + y)*(x - y));
(%o1)              lambda([x, y], (x + y) (x - y))
(%i2) f (a, b);
(%o2)                       (a - b) (b + a)
(%i3) apply (f, [p, q]);
(%o3)                       (p - q) (q + p)
(%i4) map (f, [1, 2, 3], [a, b, c]);
(%o4)  [(1 - a) (a + 1), (2 - b) (b + 2), (3 - c) (c + 3)]
```

6 Built-in object types

An object is represented as an expression. Like other expressions, an object comprises an operator and its arguments.

The most important built-in object types are lists, matrices, and sets.

6.1 Lists

1. A list is indicated like this: $[a, b, c]$.
2. If L is a list, $L[i]$ is its i'th element. $L[1]$ is the first element.
3. **map**(f, L) applies f to each element of L.
4. **apply**$("+", L)$ is the sum of the elements of L.
5. **for** x **in** L **do** *expr* evaluates *expr* for each element of L.
6. **length**(L) is the number of elements in L.

6.2 Matrices

1. A matrix is defined like this: **matrix**$(L_1, \ldots, L_n)$ where $L_1, \ldots, L_n$ are lists which represent the rows of the matrix.
2. If M is a matrix, $M[i, j]$ or $M[i][j]$ is its (i, j)'th element. $M[1, 1]$ is the element at the upper left corner.
3. The operator . represents noncommutative multiplication. $M.L$, $L.M$, and $M.N$ are noncommutative products, where L is a list and M and N are matrices.
4. **transpose**(M) is the transpose of M.

5. **eigenvalues**(M) returns the eigenvalues of M.
6. **eigenvectors**(M) returns the eigenvectors of M.
7. **length**(M) returns the number of rows of M.
8. **length**(**transpose**(M)) returns the number of columns of M.

6.3 Sets

1. Maxima understands explicitly-defined finite sets. Sets are not the same as lists; an explicit conversion is needed to change one into the other.
2. A set is specified like this: **set**($a, b, c, \ldots$) where the set elements are $a, b, c, \ldots$.
3. **union**(A, B) is the union of the sets A and B.
4. **intersection**(A, B) is the intersection of the sets A and B.
5. **cardinality**(A) is the number of elements in the set A.

7 How to...

7.1 Define a function

1. The operator := defines a function, quoting the function body.

   In this example, **diff** is reevaluated every time the function is called. The argument is substituted for x and the resulting expression is evaluated. When the argument is something other than a symbol, that causes an error: for **foo**(1) Maxima attempts to evaluate **diff**(**sin**$(1)^2, 1)$.

```
(%i1) foo (x) := diff (sin(x)^2, x);
                                   2
(%o1)               foo(x) := diff(sin (x), x)
(%i2) foo (u);
(%o2)                      2 cos(u) sin(u)
(%i3) foo (1);
Non-variable 2nd argument to diff:
1
#0: foo(x=1)
 -- an error.
```

2. **define** defines a function, evaluating the function body.

   In this example, **diff** is evaluated only once (when the function is defined). **foo**(1) is OK now.

```
(%i1) define (foo (x), diff (sin(x)^2, x));
(%o1)                  foo(x) := 2 cos(x) sin(x)
(%i2) foo (u);
(%o2)                         2 cos(u) sin(u)
(%i3) foo (1);
(%o3)                         2 cos(1) sin(1)
```

7.2 Solve an equation

```
(%i1) eq_1: a * x + b * y + z = %pi;
(%o1)                     z + b y + a x = %pi
(%i2) eq_2: z - 5*y + x = 0;
(%o2)                       z - 5 y + x = 0
(%i3) s: solve ([eq_1, eq_2], [x, z]);
                   (b + 5) y - %pi      (b + 5 a) y - %pi
(%o3)     [[x = - ---------------, z = -----------------]]
                        a - 1                a - 1
(%i4) length (s);
(%o4)                                1
(%i5) [subst (s[1], eq_1), subst (s[1], eq_2)];
        (b + 5 a) y - %pi   a ((b + 5) y - %pi)
(%o5) [----------------- - ------------------- + b y = %pi,
              a - 1                a - 1
                     (b + 5 a) y - %pi   (b + 5) y - %pi
                     ----------------- - --------------- - 5 y = 0]
                           a - 1               a - 1
(%i6) ratsimp (%);
(%o6)                       [%pi = %pi, 0 = 0]
```

7.3 Integrate and differentiate

integrate computes definite and indefinite integrals.

```
(%i1) integrate (1/(1 + x), x, 0, 1);
(%o1)                             log(2)
(%i2) integrate (exp(-u) * sin(u), u, 0, inf);
```

```
                                   1
(%o2)                              -
                                   2
(%i3) assume (a > 0);
(%o3)                           [a > 0]
(%i4) integrate (1/(1 + x), x, 0, a);
(%o4)                         log(a + 1)
(%i5) integrate (exp(-a*u) * sin(a*u), u, 0, inf);
                                   1
(%o5)                             ---
                                  2 a
(%i6) integrate (exp (sin (t)), t, 0, %pi);
                           %pi
                          /
                          [      sin(t)
(%o6)                     I    %e       dt
                          ]
                          /
                           0
(%i7) 'integrate (exp(-u) * sin(u), u, 0, inf);
                         inf
                        /
                        [      - u
(%o7)                   I    %e    sin(u) du
                        ]
                        /
                         0
```

diff computes derivatives.

```
(%i1) diff (sin (y*x));
(%o1)          x cos(x y) del(y) + y cos(x y) del(x)
(%i2) diff (sin (y*x), x);
(%o2)                         y cos(x y)
(%i3) diff (sin (y*x), y);
(%o3)                         x cos(x y)
(%i4) diff (sin (y*x), x, 2);
                                2
(%o4)                        - y  sin(x y)
(%i5) 'diff (sin (y*x), x, 2);
                             2
                            d
```

```
(%o5)                          --- (sin(x y))
                                 2
                               dx
```

7.4 Make a plot

plot2d draws two-dimensional graphs.

```
(%i1) plot2d (exp(-u) * sin(u), [u, 0, 2*%pi]);
(%o1)
(%i2) plot2d ([exp(-u), exp(-u) * sin(u)], [u, 0, 2*%pi]);
(%o2)
(%i3) xx: makelist (i/2.5, i, 1, 10);
(%o3)  [0.4, 0.8, 1.2, 1.6, 2.0, 2.4, 2.8, 3.2, 3.6, 4.0]
(%i4) yy: map (lambda ([x], exp(-x) * sin(x)), xx);
(%o4) [0.261034921143457, 0.322328869227062, .2807247779692679,
.2018104299334517, .1230600248057767, .0612766372619573,
.0203706503896865, - .0023794587414574, - .0120913057698414,
- 0.013861321214153]
(%i5) plot2d ([discrete, xx, yy]);
(%o5)
(%i6) plot2d ([discrete, xx, yy], [gnuplot_curve_styles, ["with points"]]);
(%o6)
```

See also **plot3d**.

7.5 Save and load a file

save writes expressions to a file.

```
(%i1) a: foo - bar;
(%o1)                             foo - bar
(%i2) b: foo^2 * bar;
                                         2
(%o2)                             bar foo
(%i3) save ("my.session", a, b);
(%o3)                            my.session
(%i4) save ("my.session", all);
(%o4)                            my.session
```

load reads expressions from a file.

```
(%i1) load ("my.session");
(%o4)                          my.session
(%i5) a;
(%o5)                           foo - bar
(%i6) b;
                                       2
(%o6)                           bar foo
```

See also **stringout** and **batch**.

8 Maxima programming

There is one namespace, which contains all Maxima symbols. There is no way to create another namespace.

All variables are global unless they appear in a declaration of local variables. Functions, lambda expressions, and blocks can have local variables.

The value of a variable is whatever was assigned most recently, either by explicit assignment or by assignment of a value to a local variable in a block, function, or lambda expression. This policy is known as *dynamic scope*.

If a variable is a local variable in a function, lambda expression, or block, its value is local but its other properties (as established by **declare**) are global. The function **local** makes a variable local with respect to all properties.

By default a function definition is global, even if it appears inside a function, lambda expression, or block. **local**$(f), f(x) := \ldots$ creates a local function definition.

trace(*foo*) causes Maxima to print an message when the function *foo* is entered and exited.

Let's see some examples of Maxima programming.

1. All variables are global unless they appear in a declaration of local variables. Functions, lambda expressions, and blocks can have local variables.

   ```
   (%i1) (x: 42, y: 1729, z: foo*bar);
   ```

```
(%o1)                              bar foo
(%i2) f (x, y) := x*y*z;
(%o2)                         f(x, y) := x y z
(%i3) f (aa, bb);
(%o3)                           aa bar bb foo
(%i4) lambda ([x, z], (x - z)/y);
                                             x - z
(%o4)                       lambda([x, z], -----)
                                               y
(%i5) apply (%, [uu, vv]);
                                  uu - vv
(%o5)                             -------
                                   1729
(%i6) block ([y, z], y: 65536, [x, y, z]);
(%o6)                          [42, 65536, z]
```

2. The value of a variable is whatever was assigned most recently, either by explicit assignment or by assignment of a value to a local variable.

```
(%i1) foo (y) := x - y;
(%o1)                          foo(y) := x - y
(%i2) x: 1729;
(%o2)                               1729
(%i3) foo (%pi);
(%o3)                            1729 - %pi
(%i4) bar (x) := foo (%e);
(%o4)                         bar(x) := foo(%e)
(%i5) bar (42);
(%o5)                              42 - %e
```

9 Lisp and Maxima

The construct **:lisp** *expr* tells the Lisp interpreter to evaluate *expr*. This construct is recognized at the input prompt and in files processed by **batch**, but not by **load**.

The Maxima symbol **foo** corresponds to the Lisp symbol $foo, and the Lisp symbol foo corresponds to the Maxima symbol ?**foo**.

:lisp (**defun** $foo (a) (...)) defines a Lisp function foo which evaluates its arguments. From Maxima, the function is called as **foo**(a).

:lisp (**defmspec** \$foo (e) (...)) defines a Lisp function **foo** which quotes its arguments. From Maxima, the function is called as **foo**(a). The arguments of \$foo are (**cdr** e), and (**caar** e) is always \$foo itself.

From Lisp, the construct (**mfuncall** '\$foo $a_1 \ldots a_n$) calls the function **foo** defined in Maxima.

Let's reach into Lisp from Maxima and vice versa.

1. The construct **:lisp** *expr* tells the Lisp interpreter to evaluate *expr*.

```
(%i1) (aa + bb)^2;
                                        2
(%o1)                         (bb + aa)
(%i2) :lisp $%
((MEXPT SIMP) ((MPLUS SIMP) $AA $BB) 2)
```

2. **:lisp** (**defun** \$foo (a) (...)) defines a Lisp function **foo** which evaluates its arguments.

```
(%i1) :lisp (defun $foo (a b) `((mplus) ((mtimes) ,a ,b) $%pi))
$FOO
(%i1) (p: x + y, q: x - y);
(%o1)                            x - y
(%i2) foo (p, q);
(%o2)                     (x - y) (y + x) + %pi
```

3. **:lisp** (**defmspec** \$foo (e) (...)) defines a Lisp function **foo** which quotes its arguments.

```
(%i1) :lisp (defmspec $bar (e) (let ((a (cdr e))) `((mplus) ((mtimes) ,@a)
#<CLOSURE LAMBDA (E) (LET ((A (CDR E))) `((MPLUS) ((MTIMES) ,@A) $%PI))>
(%i1) bar (p, q);
(%o1)                          p q + %pi
(%i2) bar (''p, ''q);
(%o2)                          p q + %pi
```

4. From Lisp, the construct (**mfuncall** '\$foo $a_1 \ldots a_n$) calls the function **foo** defined in Maxima.

```
(%i1) blurf (x) := x^2;
```

```
(%o1)                        blurf(x) := x
(%i2) :lisp (displa (mfuncall '$blurf '((mplus) $grotz $mumble)))
                  2
(mumble + grotz)
NIL
```